DOCUMENT RESUME

ED 097 915 IR 001 294

AUTHOR	Nievergelt, Jurg; And Others
TITLE	ACSES, An Automated Computer Science Education System.
INSTITUTION	Illinois Univ., Urbana. Dept. of Computer Science.
SPONS AGENCY	National Science Foundation, Washington, D.C.
PUB DATE	1 Nov 74
NOTE	15p.
EDRS PRICE	MF-$0.75 HC-$1.50 PLUS POSTAGE
DESCRIPTORS	College Students; *Computer Assisted Instruction; Computers; *Computer Science; *Computer Science Education; Individualized Instruction; Program Descriptions
IDENTIFIERS	*PLATO IV; University of Illinois

ABSTRACT

A project to accommodate the large and increasing enrollment in introductory computer science courses by automating them with a subsystem for computer science instruction on the PLATO IV Computer-Based Education system at the University of Illinois was started. The subsystem was intended to be used for supplementary instruction at the University of Illinois, but was also designed to be usable as a completely self-contained system for introductory computer science education for remote sites. The key components of the system are: a library of lessons, covering several programing languages, computing techniques, and application areas; a completely self-contained interactive program system for the preparation, execution, and debugging of programs written by students in any of the languages covered by the lessons; a conversational advice-giving and information retrieval system to guide the student through the library of lessons, based on his goals and past performance; and a communication system that allows a student to contact a human tutor. (Author)

ED 097915

ACSES, AN AUTOMATED COMPUTER SCIENCE EDUCATION SYSTEM

J. Nievergelt
T. R. Wilcox
H. G. Friedman
R. G. Montanelli
J. A. Barber
R. Barnett
B. Z. Barta
R. L. Danielson
A. M. Davis
D. R. Eland
D. W. Embley
W. J. Hansen
F. Izquierdo
P. Mateti
J. M. Milner
J. L. Pradels
E. M. Reingold
J. Renshaw
B. Z. Segal
M. H. Tindall
D. S. Watanabe
L. G. Whitlock

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801, USA

November 1, 1974

U.S. DEPARTMENT OF HEALTH, EDUCATION & WELFARE
NATIONAL INSTITUTE OF EDUCATION
THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

Abstract

In response to a large (currently 2000 students per semester) and increasing enrollment in introductory computer science courses, we have started a project to automate these courses by developing a subsystem for computer science instruction on the PLATO IV Computer-Based Education system at the University of Illinois. It is intended to be used for supplementary instruction at our university, but is designed to be usable as a completely self-contained system for introductory computer science education, in order to be usable at remote sites.

The key components of this system are:

1. a library of lessons, covering several programming languages, computing techniques, and application areas
2. a completely self-contained interactive programming system for the preparation, execution and debugging of programs written by students in any of the languages covered by the lessons
3. a conversational advice-giving and information-retrieval system to guide the student through the library of lessons, based on his goals and past performance
4. a communication system that allows a student to contact a human tutor.

The goals and current status of this project are described.

BEST COPY AVAILABLE

IR001 294

1. INTRODUCTION

Computers are playing an increasingly pervasive role in our society, and this fact leads directly to a rapidly increasing demand for basic computer science education. This demand arises from two sources.

First, the demand for computer professionals continues to grow. The most widely accepted projections show a doubling of the total demand for systems analysts, programmers, computer operators and associated technicians during the next five years.

Secondly, it seems reasonable to expect that most people will be required to interact with computers in their daily work within a decade or two. Even people not directly concerned with computers should have some understanding of them because an enlightened public will be essential if we are to make intelligent decisions concerning the future role of computers in our society. Currently many citizens view computers with indifference, while others fear them as the ultimate threat to their privacy, security, and dignity. Such attitudes will clearly not suffice. Hence, it is important that every educated person have some understanding of the principles underlying computers and their implications for society, as well as some skill in their use. At the very least, everyone should have the opportunity to acquire this knowledge in a convenient way.

Recognizing the importance of "computer literacy," the President's Science Advisory Committee recommended in their 1967 report that 75% of all college students should have a meaningful exposure to computers. Even though this percentage has not yet been attained, the demand for instruction in basic computer science at our universities, colleges, junior colleges, and private electronic data processing schools is enormous. While we lack accurate current figures, we can perceive the general trend by examining the data for the years 1965-1969 compiled by the Computer Sciences Project of the Southern Regional Education Board.[1] In 1964-1965, there were approximately 4,000 undergraduate and 1,300 graduate majors in computer science, data processing, information science, and related programs. By 1966-1967, these totals had jumped to 22,000 undergraduate and 5,000 graduate majors. In addition to those in degree programs, nearly 120,000 undergraduate and 29,000 graduate students received some computer training during 1966-1967. By 1968-1969, these totals had nearly tripled to 350,000 undergraduate and 81,000 graduate students. The data also shows that most of the students, facilities, and funding are concentrated in relatively few universities and colleges. The smaller colleges and junior colleges, however, are interested in providing degree programs in computer studies. The American Association of Junior Colleges has stated in its report "The Computer and the Junior College"[2] that "The community college must accept the challenge and help fill the need...(for experienced practitioners)...through sound technical education programs in business data processing." However, the financial problems attendant to such programs have prevented many junior colleges from entering this field.

These institutions have relied on the traditional instructional approach of lecture-discussion-laboratory. This approach suffers from

---

[1] "Computers in Higher Education," Southern Regional Education Board, Atlanta, Georgia, August 1967.

[2] "The Computer and the Junior College," American Association of Junior Colleges, Washington, D.C., 1969.

several defects, particularly when it involves large numbers of students. It is not particularly suited to the subject and is, therefore, the cause of student dissatisfaction. Learning to program requires active participation and intense effort on the part of the student--two things that are not encouraged by the lecture type of instruction. An individual tutor for every student would be ideal for learning a skill such as programming, but such a mode of instruction is obviously economically not feasible when one aims at a mass-education program.

In addition, the traditional lecture-based approach can only reach a limited audience. In particular, it excludes all people whose professional duties prevent them from attending school for any length of time. It is to be expected that there will be a large demand for basic computer science courses in connection with continuing adult education programs. The situation where somebody suddenly finds that he should know something about computers in order to remain effective on his job, will become an increasingly familiar event.

All of these long-range considerations, in addition to our own experience in teaching introductory computer science at the University of Illinois to about 2000 students of widely different backgrounds every semester, have caused us to start a large scale project to automate introductory computer science courses.

The PLATO IV system being developed by the Computer-based Education Research Laboratory at the University of Illinois gives us a unique opportunity to develop an automated course consisting of CAI-lessons about computers and of supporting software, and to try this system out on a large audience of diverse educational background. The PLATO IV system is now operational and is expected to have 1000 terminals in the spring of 1975, located at schools and colleges with very different types of student populations.

Potentially, computer-assisted instruction (CAI) has many advantages: It can provide truly individualized instruction by allowing students to study sequences of lessons tailored to their needs and at their own pace; given a suitable terminal network, it can reach a wide audience and, in the forseeable future, do so at low cost. Because of its potential cost effectiveness, CAI may become the cheapest way for schools and colleges who do not as yet offer computer science courses to institute programs in computer studies. However, these potential advantages of CAI will be realized only if much more research and a large scale development effort is carried out. We view our project as a contribution towards the goal of demonstrating the feasibility of a CAI-based approach to the problem of mass-education in basic computer science.

ACSES, our automatic computer science education system being developed on PLATO IV, is designed to be usable in two modes, according to the two purposes it is intended to serve: supplementary instruction at our own university, and main-line instruction at remote sites.

a) <u>The partially automated mode for supplementary instruction</u>

At our University an instructor will still be responsible for a course, and CAI lessons will be used in an adjunct mode. He may discuss a problem in the classroom, and then refer the students to an appropriate lesson that gives more detail, examples, or allows the student to practice or solve problems on the computer. In the

BEST COPY AVAILABLE

case of our computer science lessons, practicing and problem solving usually means that the student must write and execute a small program. He should be able to do so at the same terminal, and switch easily from lesson taking to programming, and back.

Thus, in order to operate a partially automated automated introductory computer science course, one needs primarily:

-- a library of lessons, covering several programming languages, computing techniques, and application areas
-- a completely self-contained interactive programming system for the preparation, execution and debugging of programs written by students in any of the languages covered by the lessons.

b) The fully automated mode for main-line instruction

We expect that the demand for basic computer science courses on the part of high schools, junior colleges, and continuing adult education will grow rapidly in the near future. In these settings, there may not be an instructor available who can guide the student's course of study and fill any gaps that might be present in the lesson material. Hence in this setting the system has to be usable in a fully automated mode, and we propose to make this possible primarily by providing:

-- a conversational advice-giving and information retrieval system to guide the student through the library of lessons, based on his goals and past performance
-- a communication system that allows a student to contact a human tutor for help and advice, from any terminal connected to the PLATO system.

We consider our project to be a major research effort to investigate the extent to which a large introductory course can be automated. We believe the time is ripe to push instructional technology to the level where it can replace an entire course or set of related courses. We also believe that if the necessary ingenuity and effort are put into the design of an instructional system and its supporting software, such a system can be made to provide an educational experience superior to the one a student has in a large introductory course taught in the conventional lecture-based manner. The effort involved in producing such a system is large. The implementation of ACES contains close to a million words of code and data, and we estimate that more than 30 man-years have been invested in it so far.

The remainder of this paper describes in more detail the instructional system we are developing, its current status, and the experiences we have gathered during preliminary use of the system over the past two years.

2. THE LIBRARY OF LESSONS

Our introductory computer science courses are offered to students of widely varying backgrounds and interests, and so these courses differ significantly from each other; for example, the problems discussed in computer courses for engineers differ greatly from those presented to business students or to teachers. Moreover, an increasing number of students have been exposed to computing in high school, hence we must cater to people with a fair degree of computer expertise, as well as to the uninitiated. On the other hand, these courses overlap a great deal when they discuss the basic structures inherent in high-level programming languages. For these reasons, we have decided against writing fixed sequences of lessons, one for each of the several courses (see, e.g.).[3] Instead, we have developed a library of about 100 lessons, from which each instructor of an introductory computer science course can select a suitable subset.

Every attempt has been made to prevent the lessons from being "page turners" which, every frame or two, ask multiple choice questions. The PLATO IV terminal is a 512 by 512 raster of individually controlled points, and as such it can easily present two-dimensional material with some, albeit slow, movement. Our philosophy in the design of the lessons is that they should take advantage of this powerful input/output device and use it to animate the material being presented.

The approximately one hundred lessons of our system can generally be divided into three categories: introductory lessons which require no prerequisites, lessons which describe the syntax and semantics of various programming languages, and lessons which describe applications of computers. From the lessons in these three broad categories it is hoped that virtually any instructor can find a large, useful subset.

The introductory lessons usually attempt to present some fundamental concept in a simplified setting. For example, the lesson called Algolingo introduces the student to the concept of an algorithm. In Algolingo, a method of finding the highest score in a set of exam scores is considered, and the "straightforward" algorithm is described in various "languages." The algorithm is first discussed in some detail and an English prose description is presented, followed by an algorithm expressed in rhyme. Finally, the algorithm is shown as very simple programs in FORTRAN, PL/1, and BASIC, with each statement carefully annotated to explain its purpose.

Frequently, the "fundamental concept" is introduced via a "mini-programming language"[4] which has been carefully designed to highlight that one idea without extraneous topics. A typical example of one such language we have designed is DOODLE (DO-loops Only Drawing LanguagE), a language for introducing the concept of loops. DOODLE has only four types of statements, and line erasing statements. For example, the program

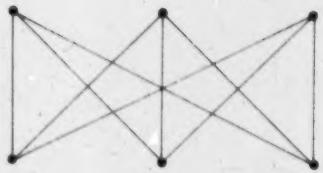
```
do a ← 1, 3
do b ← 1, 3
draw 40+40a,100; 40+40b,500;
end
end
```

[3] Fenichel, R. R., J. Weizenbaum and J. C. Yochelson, "A program to teach programming," Comm. ACM 13 (1970), 141-146.

[4] Ledgard, H. F., "Ten mini-languages: A study of Topical Issues in Programming Languages," Computing Surveys 3 (1971), 115-146.

BEST COPY AVAILABLE

produces the figure



Students write and edit DOODLE programs interactively and can then execute them with the results shown on the screen in front of them. Despite its simple structure, DOODLE can be used to produce quite sophisticated drawings.

Typical of the sequences of lessons for various programming languages is the PL/1 sequence, which consists of about a dozen lessons on various facets of the language. There are lessons for introducing the data types of PL/1, the operations on those data types, IF statements, DO statements, input/output, and procedures. Finally, there is an examination to test the students' proficiency with PL/1. A similar sequence exists for FORTRAN; others have been started for BASIC, COBOL, SNOBOL, and APL.

The lessons on various applications are quite diverse, covering topics such as business data processing, simulation, and numerical analysis. For example, an introductory lesson on file processing is followed by a more advanced lesson on sorting, which discusses various methods such as insertion sorting, bubble sorting, and quicksort. Students are asked to give sequences of numbers to be sorted and they are asked, in turn, to sort sequences given to them.

The numerical analysis sequence currently consists of lessons on least squares approximation, numerical quadrature, the solution of a single nonlinear equation, the solution of linear systems of equations, and linear programming. PLATO IV is ideally suited to the teaching of numerical analysis because of its graphical and interactive computing capabilities. For example, the ideas underlying root finding methods become transparent when stated geometrically and presented graphically, and the notion of iteration becomes obvious to a student observing the action of typical root-finding schemes applied to functions chosen by him. The graphical capabilities further provide means of motivating the development of numerical algorithms. For example, in the least squares lesson the student first performs an experiment where he measures the extention of a "rubber band" caused by various weights. Then faced with the problem of fitting a straight line to his data, he is introduced to least squares approximation.

The lesson on Monte Carlo techniques discusses how random behavior can be used to obtain numerical solutions to deterministic problems such as finding areas under curves and solving potential equations. The student can draw any polygonal figure and then observe as its area is computed by selecting points at random and calculating the fraction found to be inside the polygon. Similarly, the student can specify any polygonal region and boundary conditions and then observe as random walks are used to find an approximation to the temperature at an internal point chosen by the student.

Other lessons are available for the students to use in which the main idea is to convince the students that computer programming can be

enjoyable as well as challenging. For example, in a lesson called SOMAGA (SOftware MAnagement GAme), students compete among one another in trying to write a DOODLE program to produce a rather sophisticated figure. The students bid competitively for the task and then must write the program in accordance with the time constraints in their bids. In another lesson, currently being developed, students write programs to play a game called Darwin[5] in which their programs try to reproduce themselves and to destroy opposing programs. These lessons use the fact that PLATO terminals can communicate with each other.

[5] R. (pseudonym) "Computer Recreations: Darwin," Software-Practice and Experience 2 (1972), 93-96.

BEST COPY AVAILABLE

3. INTERACTIVE PROGRAMMING SYSTEM

After a student has been presented with an idea, he should have an opportunity to practice it immediately. This means that he should be able to enter and execute programs from the same terminal where he is following the lessons. These need not be large and time-consuming programs; much programming can be learned with 20-line programs and execution times of a fraction of one second.

On the other hand, the quality of diagnostics is of the utmost importance, since in an automated programming course, the programming system must also take over the duties of the consultant who usually assists students in a traditional programming course. This means that the interactive compiler in a computer aided programming course must be adept in explaining errors to students. Wherever possible, it interacts with the student in his own terms, in English. Thus, to be effective, an automated consultant must combine the best features of interactive systems and 'student-' or 'debugging-' compilers into one program.

This section discusses the features of a completely self-contained interactive programming system, referred to as the "compiler/consultant", for the preparation, execution and debugging of programs written in any of several current high-level languages. The features which make the system particularly suited for use in a computer aided programming course are

a) the ability to accept a user-defined subset of each language,

b) the flexible text preparation facility,

c) on-line reference materials,

d) a short-term filing system, and,

e) extensive diagnostic assistance.

This programming system currently supports subsets of PL/1, FORTRAN, and COBOL suitable for a beginners programming course, and work is almost complete on BASIC. Since the system is largely table driven, adaptation to other languages is straightforward.

a) User-defined subsets

It is not generally the language itself that makes a full-featured language like PL/1 difficult to teach. The blame usually rests with the compiler for the language. The established way to teach a large language is to present it feature by feature and each feature, in turn, in more and more detail. If, however, at any point in this process the compiler does not restrict the student to the features he has learned, it is all too easy for the student to stumble onto an unknown region of the language. Such ventures will most likely generate compilation errors which are keyed to the unknown feature and therefore a source of confusion to the student. For example, what is a student to think if in his first PL/1 program he is told he has a missing ON condition when he wrote 'ON N = 1;' instead of 'ON: N = 1;'? In the rare event the errant construct is meaningful to the compiler, the program will not execute as expected and the programmer will be at a loss for an explanation.

To avoid such problems, the compiler/consultant is designed to assume that the student is using only the subset of the language to which he has been introduced. As the student completes a lesson, flags in his continuing record are set indicating the language features the lesson has covered. These flags may also be set by a student directly if he has acquired knowledge of a feature outside the system of lessons. The feature flags are used primarily to establish the initial symbol table configuration and thus define the subset the compiler/consultant should assume (it cannot recognize a DO-statement if DO is not present in the symbol table as a 'statement keyword'). The flags are also used to modify the text of certain error messages and the compiler/consultant's response to the student's questions.

This subsetting feature also permits lessons in the system to use the compiler/consultant to parse and execute selected language features. A lesson drilling on the use of IF-statements, for example, can enter the consultant with flags set to accept only IF-THEN statements. The compiler/consultant will read and analyze the student's input and pass the internal representation back to the lesson for further analysis. Later, this internal representation can be executed by using another entry point to the compiler/consultant. This technique avoids any duplication of parsing and execution effort between the lessons and the compiler/consultant.

b) Program Entry and Editing

In addition to learning a language, the student must also learn how to prepare his program at the terminal and then put it into execution. Program entry and editing is designed to be as natural as possible. There are no editing commands to speak of. The situation the system tries to simulate is as if the student were sitting at a typewriter with an eraser. He is able to move the carriage (indicated by a triangular cursor on the screen) up and down and left and right by the touch of a key. Wherever the cursor is sitting, he is able to erase existing characters or add new ones. Entire lines may be inserted or deleted by simple key presses. In addition to being easy to learn and use, this technique eliminates the

need for line numbers, thus leaving more room on the screen for program text.

In an effort to teach good programming style, the consultant automatically formats the program on the screen as it is being entered -- in PL/1, semicolons terminate a line, BEGIN and PROCEDURE blocks, DO-loops and nested IF's are indented and labels are justified to the left margin. This format can be changed -- to squeeze several short statements onto one line, for example -- by reediting the text after it has been entered.

c) On-line Reference

Selected reference materials are always available on-line for immediate consultation. These include

1) a brief description of the syntax and semantics of the language
2) a list of the built-in functions available,
3) instructions on how to type special characters,
4) a summary of the editing keys and their function,
5) the collating sequence, and
6) a list of the variables defined in the current procedure and their attributes.

d) On-line Storage

The compiler/consultant also provides temporary on-line storage for programs and data. Program material is saved as external procedures under the procedure name. Data is saved as line files (64 characters per record) and may be referenced in a program via the standard I/O statements of the language. This facility is provided for short-term storage (2-3 days) of student programs in preparation and long-term storage of grading programs and data and other materials a course instructor wishes to share

with his students. Each course using the consultant has its own filing system. There is a one-level name space within each filing system to permit sharing of grading programs and data. Files are password protected to prevent unauthorized copying of solutions.

e) Diagnostic Assistance

Program Preparation Errors. Syntax and semantic analysis is performed character by character as the student enters is program. Even in the editing process the compiler/consultant stays with the student watching every character being erased or entered and ready to help at the first input which puts the program outside the student's subset. This technique is used for two reasons. First, immediate reporting of errors means that the student will have a better chance of understanding the error message the first time it is presented. The student sees exactly what the consultant has seen. Even having no knowledge of how a compiler goes about analyzing its input, the student will know what triggered the error message. It's always the last character he typed. There is also the possibility that the error message will have greater psychological impact, coming as it does at the very point of experimentation.

Issuing the error message at the point the offending character is entered should also reduce the frustration of entering a long or complicated line only to find a comma is missing halfway through the line. Of course, by using just a few editing keys the comma can easily be inserted without retyping the line, yet extra key presses are still needed to properly repair the error.

On the other hand, immediate and consistent complaint on the slightest error could itself be frustrating to the student, especially if

BEST COPY AVAILABLE

he is not a proficient typist! In an attempt to avoid this, the compiler/consultant issues an error message gradually with interaction from the student.

When an error is detected, a box is drawn around the offending character or token. If the student understands the mistake, he simply corrects the error using the normal editing keys. If he doesn't understand the error, he presses a key marked 'HELP.' This causes the compiler/consultant to write an informative message on the screen. If the student still does not understand he may press HELP again and this time the text is marked to show the nature of the error. If, for example, he has unbalanced parentheses, lines are drawn to show how the parentheses match and which parentheses do not match. If a label or variable is multiply defined, the conflicting definition is circled. The terminal has elaborate graphics capabilities which the compiler/consultant uses heavily to explain errors.

Program Execution Errors. The level of error diagnostics and sophisticated interaction cannot let up during execution of the student's program. In the compiler/consultant, the same type of interactive discussion of error situations exists at run-time as discussed above for program preparation. The final response in a help sequence in this case is a list of suggestions on what could be wrong and what to look for. These suggestions are compiled on the basis of the error and additional execution statistics such as the number of instructions executed or the current procedure or DO-nesting structure.

Execution errors are presented by displaying the text of the procedure containing the error on the screen and marking the specific point of the error. An error message is also displayed.

To aid the student in investigating execution errors, extensive debugging tools are always available. These include the ability to inspect or modify the value of any of his variables and to step through a particular section of program. Four step sizes are available: entire statements, phrases of statements, expressions, and single operations. In addition, the student can execute the program backwards, statement by statement, in order to track down the source which ultimately gave rise to an execution error detected by the system (the entire history of program execution is kept for this purpose). At any moment, the student may also go back and edit his procedures and then reexecute his program.

4. A CONVERSATIONAL INFORMATION-RETRIEVAL SYSTEM

When a student approaches a terminal, how can he determine which lesson he should take? The full potential of CAI for individualized instruction will not be achieved by some fixed scheduling technique. In our automated course we propose to leave the choice of lesson to the student, but we must provide him with accurate information on which he can base his decision. To this end, we have developed a program called the "Guide," which is a conversational information-retrieval and advising system. Here is a sample of requests to which the Guide gives appropriate responses.

"What are the prerequisites of lesson Fortdo"

"Which PL/1 lessons must I take for CS 101"

"A lesson similar to the one I took yesterday"

"What grade did I get in the last exam"

"What do you have on bubble-sort"

The Guide responds by displaying a list of lesson names, or a lesson record (lesson name, author, abstract, indexing keywords), or a course outline, or a student record, or an appropriate part of these.

In order to be able to process such requests, the Guide has access to the following four data-bases:

a) A library catalog describing the content of each lesson as well as the relationships among lessons.

b) A subject classification scheme that covers all the topics treated in any of the courses.

c) A record of the lessons each student has taken, and of his performance.

d) A course outline and schedule set up by an instructor.

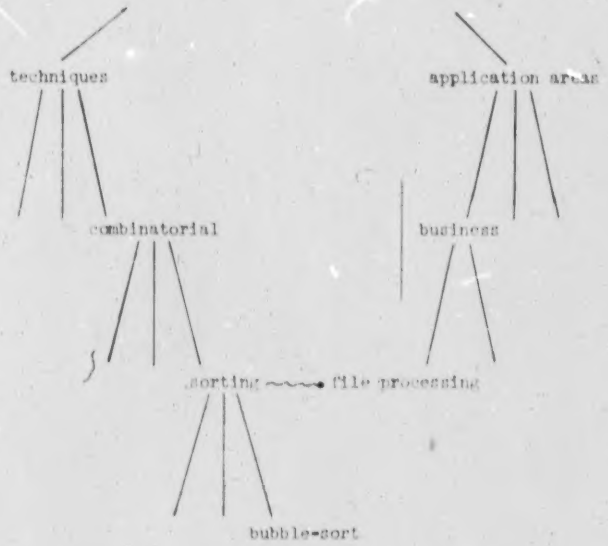
Let us briefly analyze the process whereby the answer to the last request can be derived. If there are any lessons indexed by the keyword "bubble-sort," they are retrieved directly. However, since "bubble-sort" is a highly specific concept, there happen to be no lessons in our library with that keyword. Instead, the catalog contains the following two records:

a) Lesson name: sort
Keywords: sorting, algorithms

b) Lesson name: files
Keywords: file, record, file processing.

This example is chosen to illustrate the use of the most interesting data base available to the guide, the subject classification scheme or "concept space." While the catalog shows what subjects are currently covered by lessons, and hence is highly time-dependent, the concept space classifies all the concepts and topics that tend to occur in introductory computer science courses (regardless of whether or not there happen to be lessons on these topics) and hence is constant in time (over the life-span of our project). It is a generalization of the Dewey Decimal system used widely in libraries, but is not restricted to a tree-structure. The portion of the concept-space that enables the Guide to present the two lessons "sort" and "files" in response to a request for something on "bubble-sort," even though "bubble-sort" does not occur as a keyword in either lesson, is shown below.

BEST COPY AVAILABLE



After the Guide has searched unsuccessfully for lessons indexed by the keyword "bubble-sort," it expands the user's original request to include keywords which are close to "bubble-sort" in the concept space. In the example, this expansion leads first to the keyword "sorting," the immediate predecessor of "bubble-sort" in the classification tree underlying the concept space. The expansion continues along a link representing the relationship "x is an important concept for y," to include the keyword "file processing." The keywords "sorting" and "file processing," which were not in the original request, now permit the retrieval of the lessons "sort" and "files."

This example illustrates the operation of one part of the Guide, the request processor. It is the task of another main part, the translator, to translate requests in English into a search prescription, expressed in an internal language, for the request processor. Because of the limited domain of discourse of the Guide, the translator is quite simple: a finite-state transition diagram with a vocabulary of a few thousand words.

5. JUDGING PROGRAMS

A lesson, after having presented some new topic or technique, frequently will ask the student to write a program which uses the material just presented. The teaching of introductory CS courses can be completely automated only if the system itself can evaluate these student programs. The system should not simply check whether the program gets the correct answer; instead it must be able to analyze the program, to spot inefficiencies and other instances of poor programming practice, and to give the student constructive criticism of his program.

Judging a program automatically in such a way as to give the student helpful information is an open-ended problem, to which many of the techniques developed in artificial intelligence research and in proving the correctness of programs are applicable.

Approaches to the automatic judging of the quality, or elegance, of programs differ on two scales: generality, i.e., the size of the class of problems to which they can be applied, and perceptiveness, i.e., the degree to which they are able to identify the real cause of why a particular program is ill-conceived or ill-structured.

At one extreme there is a technique that is specific to each particular programming problem, but can be expected to give comments comparable in quality to those of a human consultant. As a test case we have developed a program to judge student-written programs for symbolic differentiation of arithmetic expressions. We are attempting to represent in one graph all "reasonable" approaches to this program, as well as a few plausible "wrong" approaches. A node in this graph represents

the progress a student has made in conceiving his program, as well as certain major decisions he has made on how to approach the problem. We ask "How do you intend to proceed now?" and hope to have a meaningful comment for every plausible course of action that the student might choose at that point. The main difficulty with this approach is that for even simple programming problems the space of reasonable approaches is large. We are currently experimenting with an AND/OR graph of 50 nodes.

A second approach has more generality but leads to less perceptive judging programs. We use as a test case the class of all programs that are supposed to sort a linear array of numbers. We provide a programming language and an assertion language, both of which are highly specific to sorting. The student is supposed to write a sorting program, with assertions about the state of the array interspersed in his program. The judging program analyzes the student's program with its assertions as to their logical consistency as well as economy. In this approach we can draw upon many results that have been obtained recently about proving the correctness of programs.

BEST COPY AVAILABLE

6. COMMUNICATION FACILITIES

In using the instructional system described, as well as in creating it, several kinds of communication become important. The following communication facilities are provided:

a) On-line assistance to students by a human consultant.

A student in a computer science lesson can contact a human consultant by pressing a special key. During times when computer science students are scheduled to be using PLATO, a human consultant is stationed at a terminal to respond to individual requests for help.

b) Messages between students, instructors, and the management.

As students take PLATO lessons, they of course develop opinions about the method of presentation, clarity of the lesson, usefulness of the exercises, etc. And they sometimes explore the lesson in ways the author did not anticipate, discovering problems of presentation and outright bugs in the code.

In order to encourage students to report problems immediately, each lesson implements a comment feature. The student activates this feature by pressing a special key and typing the word "comment." In response, a program called "cscomments" is activated. The student then enters his comment on the lesson he was studying. He can also read comments others have made, both about this and about other lessons. When finished, he returns to the lesson he was taking, and continues from the point at which he left off. The management looks at these collected comments periodically, enters responses into the comment file, or takes corrective action.

An instructor can write messages to any or all of his students. These automatically appear on the student's screen next time he signs on.

7. EXPERIENCE WITH THE SYSTEM

Our system has been used in an experimental way, involving a total of a few hundred students, for the past two years. In order to evaluate the overall effect of the use of our instructional system in a course, we have carried out two controlled experiments.

In the fall of 1973, we carried out a two-month experiment with 120 students in an introductory course for computer science majors. Half of these students learned PL/1 from a sequence of 8 PLATO lessons, while the other half learned PL/1 by attending a 1-hour discussion section per week. These were given by graduate student teaching assistants to groups of 20 students at a time.

A questionnaire was administered to assess students' attitudes toward PLATO. Sixty-two percent (38/61) of the students disagreed or strongly disagreed with the statement "Computer-based education is nothing but an expensive gimmick," while 23% were uncertain or omitted the question and 15% agreed or strongly agreed. Similarly, 64% disagreed with the statement "Computer-based education dehumanizes the student" with 21% uncertain and 15% agreeing. Finally, when asked to give advice to a friend as to whether to take a PLATO section or not, 33% advised to take a PLATO section, 46% were neutral (or didn't answer), and 21% advised against. On balance, it would seem that students reacted favorably to PLATO, even though the lessons were not perfect.

Finally, a written PL/1 exam was given to all students, and there were no statistically significant differences between PLATO and non-PLATO groups.

This experiment was considered mainly to be a trial run to debug our experimental procedures. In the fall of 1974 it has been repeated with a class of 60 students, half of whom learned FORTRAN from a sequence of 12 PLATO lessons, while the other half attended 12 one-hour lectures on FORTRAN. The data from this experiment has not yet been analyzed, but our initial impression is that the two groups performed about equally well on a FORTRAN test, and that the satisfaction of the PLATO group was significantly better than it was one year ago in the first experiment. We consider this to be a confirmation that our instructional system has improved significantly, and is now ready for routine use in large courses. Its first test with large numbers of students (approximately 600) is planned for the spring semester 1975.

Acknowledgment

We are grateful to the Computer-based Education Research Laboratory at the University of Illinois (Professor D. L. Bitzer, Director) for providing the facility which made our instructional system possible: the PLATO IV system. Dick Blomme, Bruce Sherwood, and Paul Tenczar of CERL have given us valuable advice on the implementation of our system.

This work has been supported in part by the Department of Computer Science at the University of Illinois, and in part by the National Science Foundation under grant NSF-EC-41511.